NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

NAVAL LOGISTICS SIMULATOR

by

Anthony W. Troxell

September 1999

Thesis Advisor: Arnold H. Buss
Second Reader: Kevin J. Maher

Approved for public release; distribution is unlimited

19991126 072

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE September 1999	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
NAVAL LOGISTICS SIMULATOR

5. FUNDING NUMBERS

6. AUTHOR(S)
Troxell, Anthony W.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

This thesis provides initial development of an interactive simulation for training Operational Logistics students in the management of Naval Operational Logistics. The model is designed with a modular architecture, enabling the flexibility to upgrade or modify selected components without altering the rest of the simulation. The simulation is implemented in the Java programming language, allowing the model to run on all major operating systems. The major components of the model include a discrete event simulation, a Graphical User Interface (GUI), and controller classes that connect the two. These controller classes pass user commands to the non-visual simulation for execution and information from the simulation to the GUI for display. Data required by the non-visual simulation is inputted from a separate database and configuration files. This feature allows the simulation to run different scenarios with distinct maps and graphics with no modification to the compiled computer code. The simulation and data structure developed in this thesis provide a solid foundation for further expansion into a fully featured interactive naval logistics training simulation.

14. SUBJECT TERMS
Java, Logistics, Modeling and Simulation, NAVLOGS

15. NUMBER OF PAGES
72

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL LOGISTICS SIMULATOR

Anthony W. Troxell
Lieutenant, United States Navy
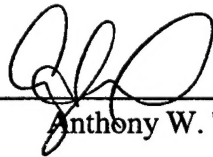B.S., United States Naval Academy, 1991

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

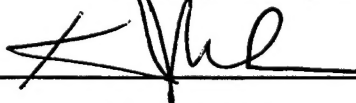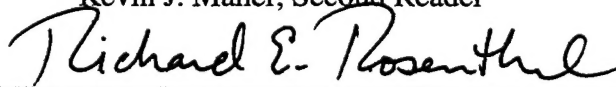## NAVAL POSTGRADUATE SCHOOL
September 1999

Author: _____
Anthony W. Troxell

Approved by: _____
Arnold H. Buss, Thesis Advisor

_____
Kevin J. Maher, Second Reader

_____
Richard E. Rosenthal, Chairman
Department of Operations Research

ABSTRACT

This thesis provides initial development of an interactive simulation for training Operational Logistics students in the management of Naval Operational Logistics. The model is designed with a modular architecture, enabling the flexibility to upgrade or modify selected components without altering the rest of the simulation. The simulation is implemented in the Java programming language, allowing the model to run on all major operating systems. The major components of the model include a discrete event simulation, a Graphical User Interface (GUI), and controller classes that connect the two. These controller classes pass user commands to the non-visual simulation for execution and information from the simulation to the GUI for display. Data required by the non-visual simulation is inputted from a separate database and configuration files. This feature allows the simulation to run different scenarios with distinct maps and graphics with no modification to the compiled computer code. The simulation and data structure developed in this thesis provide a solid foundation for further expansion into a fully featured interactive naval logistics training simulation.

THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms

AE	Ammunition Ship
ALSB	Advanced Logistics Support Base
AO	Fleet Oiler
AOE	Fast Combat Support Ship
API	Application Programming Interface
ARG	Amphibious Readiness Group
CLF	Combat Logistics Force
FLS	Forward Logistics Site
GUI	Graphical User Interface
JDBC	Java Database Connectivity
JVM	Java Virtual Machine
NAVLOGS	Naval Logistics Simulator
NPS	Naval Postgraduate School
NTDS	Naval Tactical Data System
ODBC	Object Database Connectivity
OOP	Object Oriented Programming
PC	Personal Computer
SQL	Structured Query Language
STREAM	Standard Tensioned Replenishment Alongside Method
UNREP	Underway Replenishment
VLS	Vertical Launching System

THIS PAGE INTENTIONALLY LEFT BLANK

Executive Summary

The modern carrier battle group is a dominant force for national policy makers. Its mere presence can exert strong pressure on potential adversaries. If the act of deterrence fails, it has the offensive power to quickly bring the fight to the enemy. However, without an effective operational logistics system supporting the battle group, it cannot sustain combat operations or remain in a theater for more than a few days. For the battle groups to be effective, it is important for naval officers to understand the logistics support system.

Students of the Operational Logistics curriculum at the Naval Postgraduate School are required to take course which introduces the fundamentals of the naval logistics system. As a training aid, part of the course is spent playing an interactive computer war game called PROLOG. PROLOG's single scenario models a notional battle group operating off the coast of a small, hostile nation, conducting medium intensity combat operations over the course of thirty days. As fuel and ordnance is consumed in the scenario, students use assigned logistics assets to keep the force supplied and operational. PROLOG contains very robust combat and logistics models with excellent training functionality. However, PROLOG was developed in the middle 1980's and will only run on a mainframe computer. The interface to the program is understandably dated, and saving a campaign in progress is frequently troublesome. The requirement for this thesis was prompted by these limitations.

This thesis provides a generic foundation for an interactive simulation to replace PROLOG. The simulation in this thesis is implemented in the Java programming

language. The use of Java enables the simulation to operate on the cheaper, more plentiful personal computer (PC) platform, or any other system with a Java Virtual Machine (JVM). The simulation is constructed with a modular design, allowing components to be modified or replaced with little or no change to the remaining elements. This gives the simulation greater flexibility and capability for expansion than its predecessor.

The simulation consists of a non-visual simulation connected to a graphical user interface (GUI). The units modeled in the simulation consume fuel at realistic rates and are prevented from maneuvering without sufficient fuel. The simulation also models the curvature of the earth to provide realistic interaction between the units. User speed and movement commands are passed from the GUI to the units. As the units move in the non-visual simulation, the GUI displays them to the user using standard Naval Tactical Data System (NTDS) symbology. All data and parameters required by the simulation are provided from separate configuration files and a database.

The data source used by the simulation is a key feature of this model. By using this design, the simulation is independent of the data. This enables the simulation to present different scenarios and maps to the user with no change to the simulation computer code. A scenario editor is also provided to allow the instructor to tailor scenarios to match the current state of the naval forces modeled. While this project is not yet fully implemented, this thesis provides a very expandable basis for a replacement war game.

I. INTRODUCTION

A. BACKGROUND

The modern carrier battle group is a dominant force for national policy makers. Its mere presence can exert strong pressure on potential adversaries. If the act of deterrence fails, it has the offensive power to quickly bring the fight to the enemy. However, without an effective operational logistics system supporting the battle group, it cannot sustain combat operations or remain in a theater for more than a few days. The carrier battle group and its required logistics support have gradually evolved over the course of this century to meet the changing threats to national interests around the world.

Over the history of the United States Navy, there have been steady improvements to the ships it takes to sea and war. Sailing frigates gave way to vessels with combined sail and steam power. Once the reliability of the new propulsion system was established, steam powered ironclads became the fleet standard. During this evolution in propulsion, the smoothbore, muzzle loading cannons of the sailing fleet were replaced by rifled, breech loading guns. By the turn of the century the heavily armored dreadnought battleship was the pride of the fleet. This century has seen the revolutionary change from a fleet of battleships to the carrier battle group with high performance aircraft and gas turbine propelled escorts with precision guided munitions and long-range cruise missiles.

The common thread throughout this history is the need for logistical support for the fleet. Whether coal and shells for the turn of the century battleship fleet or JP-5

1

aviation fuel and Tomahawks for today's force, combat operations cannot be sustained without an effective logistic system. Operational logistic support has progressed along side the evolution of the fleet. The modern carrier battle group is supported by the combat logistics force (CLF), a fleet of purpose-built ships designed to replenish the force at sea. Without the CLF, a carrier battle group cannot sustain combat operations or remain in a theater for more than a few days. The operations of the CLF are a complex and vital component of naval operations.

In order to successfully employ weapons and platforms of the modern fleet it is necessary to understand how the fleet is supported. The same technology which led to the development of the modern naval force has allowed advancements in training techniques. At the turn of the century, manual board games were the means of training and doctrine development. The rise of computers has enabled the manual wargame to be replaced with interactive simulations. To understand the issues for educating and training naval officers in operational logistics, it is important to understand how the modern CLF fleet and its operations have evolved. Therefore, before examining the impetus for this thesis, a brief history of naval logistics will be explored.

For most of the United States' early history, the obligations of the Navy were largely confined to the continental shores. Since the beginning of this century the country gradually become an international power with interests around the globe. Much of the burden of these new overseas commitments has fallen to the Navy. With forces

2

increasingly deployed world-wide and no guarantee of a friendly port in the theater of operations, it became necessary to bring logistic support along with the fleet.

During this time, rapid technological development further confounded the problem of supplying the Navy's ships. Coal, and later oil, had to be supplied, as well as shells for breach loaded guns. The problem of replenishment came to the forefront during the Spanish American War in 1898. The colliers dispatched to refuel the battleships blockading Cuba could not transfer at sea, necessitating the capture of Guantanamo Bay to provide a protected anchorage for re-supply. [Ref. 1: p 5] In addition to the time off station for refueling, combatants were also diverted from their primary mission to prevent the capture or destruction of the slow, vulnerable colliers. The experiences of this war were the catalyst that sparked the research for underway replenishment (UNREP) in the Navy.

While a functional system of coaling at sea was developed in the years prior to World War I, it was never fully implemented in the U.S. Navy. The ungainly topside equipment and slow rate of transfer generated considerable resistance from the senior leadership that ultimately doomed this system. Furthermore, by 1914 most Navy vessels were converted to run on oil rather than coal. Interest in underway replenishment, however, did not wane. When the U.S. entered the war in 1917, most capital ships could reach Europe without refueling; the destroyers so badly needed to combat the German U-boat menace could not. To get these destroyers to the theater, an oiler, the *USS Maumee* (AO 2), was stationed 300 miles south of Greenland. Her ingenious executive officer,

Lieutenant Chester Nimitz, designed a procedure for underway refueling at five knots. In her three months on station, *Maumee* successfully refueled thirty-four destroyers. Nimitz' procedure would become standard throughout the fleet during the interwar years. [Ref. 1: pp. 6-8]

Although the Navy had established procedures to refuel combatants and even carriers while underway by World War II, the closing months of the Pacific campaign revealed another serious deficiency in logistic support. The carrier forces attacking the Japanese islands were consuming all carried ordnance in only three to four days. When the ordnance was expended, the carriers required ten to twelve days to transit to the nearest U.S. bases, re-arm, and return to station. Under the direction of Admiral Raymond Spruance, commander of the Iwo Jima and Okinawa invasion force, a technique was devised to transfer ammunition underway by February 1945. Once this hurdle was cleared, underway replenishment groups of oilers, ammunition ships, and stores ships were stationed outside the range of Japanese kamikaze attacks. The depleted combatants would travel to the replenishment groups at night, spend the next day restocking, then transit back to resume the air strikes. This innovation allowed offensive forces to spend significantly more time attacking than replenishing. [Ref. 1: pp. 8-9]

During the rapid drawdown of forces following the war there was little innovation in underway replenishment. As a result, the procedures used during the Korean conflict were essentially the same as the Pacific operations. The carriers were fitted with additional receiving stations, reducing the time required for re-arming. However, the

4

introduction of jet aircraft with four times the fuel consumption of piston aircraft made refueling a much more difficult problem. [Ref. 1: p 10] The modern underway replenishment fleet was born out of a 1957 conference called by the Chief of Naval Operations, ADM Arleigh Burke. [Ref. 1: pp. 12-13] ADM Burke recognized from his experiences in World War II that, "[a]ll time spent in replenishing was time lost in combat." [Ref. 1: p 13] The centerpiece of CLF fleet would be the Fast Combat Support Ship (AOE).

The concept of the AOE was based on a captured German oiler, the *Dithmarschen*, that was also equipped with holds for ammunition and stores. [Ref. 1: p 11] This configuration reduced the time combatants spent off station, since they were able to replenish all commodities in one replenishment. The Standard Tensioned Replenishment Alongside Method (STREAM) equipment was designed concurrent with the planning of the AOE. The STREAM equipment enabled faster, safer transfer of cargo in more adverse weather conditions. [Ref. 1: pp. 17-18] The U.S. design also called for the AOE to be fast enough to stay with the battle group as a station ship. Rather than the World War II model of underway replenishment groups, the logistics ship would travel with the force to replenish them as required. [Ref. 1: p 57] Although the original plan called for one AOE for each deployed battle group, the Navy has never had a sufficient number of AOE's to make this possible.

In the absence of an AOE, a combination of a fleet oiler (AO) and an ammunition ship (AE) may fill the station ship role. [Ref. 1: pp. 57-58] Although the AO's and AE's

5

currently in the fleet are "purposely-built" single commodity station ships, the preferred function of these ships is to act as shuttle ships. When the station ship (or ships') supplies are expended, the station ships themselves are replenished by shuttle ships. The shuttle ships are AO's and AE's not acting as station ships, operating from advanced logistics support bases (ALSB) or less capable forward logistic sites (FLS). [Ref. 1: p 55] Both ALSB's and FLS's contain stockpiles of materiel delivered from the U.S. by efficient merchant tankers and container ships for pickup by shuttle ships. The ALSB differs from the FLS in that it also has the capability for "[s]hip and aircraft repair and maintenance, medical, personnel staging and administrative operating control centers...,". [Ref. 1: p 55]

The global landscape has seen rapid change in the past decade. During the Cold War, the Navy was faced with a single powerful opponent. Any expected combat with that opponent would be decisive, open-ocean battles. The threats of the post Cold War world are less powerful, but far less localized. To deal with these new opponents, the missions of the Navy have shifted to the coastal regions of the world. No matter what the composition of the fleet is or its theater of operations, the requirement for logistic support will survive.

B.   PROBLEM STATEMENT

While most naval officers are familiar with history and concepts outlined above, few have direct, staff level experience in coordinating the replenishment of battle group assets. All Operations Logistics students at the Naval Postgraduate School (NPS) are

required to take Introduction to Naval Logistics (OA 3610), a course designed to present the fundamentals of the naval operational logistics system. Obviously, it is impractical to give these students direct exposure to all facets of operational logistics management. The use of interactive wargames and simulations has the potential to provide cost effective, but meaningful insight to this complicated problem.

Currently, OA 3610 uses a program called PROLOG to introduce logistics management of fuel, spare parts, and ammunition requirements. While assigned as a student, LCDR Mark Mitchell originally developed PROLOG in 1982 as a manual war game. When he returned as a faculty member in 1985, he programmed a version of the game in FORTRAN for play as an interactive simulation on a mainframe computer. [Ref. 2: p ii] Subsequently, five students completed theses improving the functionality of the basic program. [Ref. 2: p ii] These theses included air strikes, air, surface, and sub-surface threat forces, player responses to these threat forces, aircraft readiness, and surface combat. The last development of the current version of PROLOG was completed in September 1988. [Ref. 2: p ii]

PROLOG presents a single battle scenario consisting of a middle 1980's notional battle group operating off the coast of "Etas Orango," a small nation embroiled in an attempted coup. Within this single scenario, PROLOG contains very robust combat and logistics models. As a training tool, the functionality provided by PROLOG is still quite good. However, ten years of advancements in hardware and software, have diminished the training value of the model. Saving and restoring a campaign in progress is

7

problematic, a significant limitation since students require about ten to fifteen hours of game play to complete the thirty day scenario. The graphics and menus of the user interface are also understandably dated. The single greatest limitation of PROLOG is the inability to update its scenario to reflect modern naval forces and missions.

C.   PURPOSE

The limitations of PROLOG noted above generated the requirement for this thesis, which develops a new simulation model, the Naval Logistics Simulator (NAVLOGS). The goal of NAVLOGS is to provide an educational tool that gives logistics students an introduction to managing fuel and ordnance at the battle group level. NAVLOGS seeks to preserve the basic functionality of PROLOG, but with more modern graphical user interface and greater flexibility and capability for expansion than the original. Other potential benefits include the ability to distribute the war game over a network of computers or even the Internet, or exporting it to other institutions.

NAVLOGS' design avoids the problems associated with the monolithic PROLOG. By utilizing a modular architecture, NAVLOGS can easily add different scenarios or update the modeled units without completely re-writing the simulation. To support the easy addition of new scenarios or units, all data required by the simulation are provided from external configuration files and a database. With this design, a new scenario or more modern units can be added by supplying updated initialization files and a new database. No change is required to the compiled computer code. While PROLOG executes only on outdated mainframe computers, NAVLOGS is designed to run on either

workstations or personal computers, which are cheaper and more plentiful. Finally, this thesis adds a "curved earth" model to discrete event simulation methodology that takes into account the curvature of the earth in mediating the interaction of targets and sensors.

Concurrently with this thesis, Lieutenant John Sterba is developing a module to add a simulation of a Marine Amphibious Readiness Group (ARG) conducting ground operations on the hostile shore. [Ref. 3] When fully implemented, these two theses will use a common interface to present future students with the logistic decisions that a commander faces when supporting a traditional Navy battle group in conjunction with ashore power projection. The underlying simulation uses an efficient discrete event model, rather than the often-used time-stepped model.

D. THESIS OUTLINE

This thesis develops a basic movement and environment model, along with a means of data storage and retrieval to support the model. While not yet completely operational, when connected with the work of Lieutenant Sterba, NAVLOGS will provide a solid foundation for a fully featured replacement for PROLOG. The remainder of this thesis is organized as follows: Chapter II provides a description of the model; Chapter III details the input of data to the model, as well as the output from the model; finally, Chapter IV provides a summary and recommendations for further expansion.

THIS PAGE INTENTIONALLY LEFT BLANK

II. MODEL METHODOLOGY

Before reading this chapter, the reader may wish to refer to Appendix A, Object Oriented Programming and Java.

A. NAVLOGS OVERVIEW

The model consists of four major components: a non-visual discrete event simulation, a graphical user interface (GUI), and controller classes that connect the two. Parameters required to create the entities in the simulation are provided from configuration files and an independent database, described in the Chapter III. The simulation is implemented in the Java programming language making the model platform independent. Currently, the Navy (and the Naval Postgraduate School) uses a mix of Windows and UNIX computers. Should this composition change in the future, NAVLOGS will still be able to execute, since it runs on any platform with a Java Virtual Machine (JVM).

The discrete event simulation logic is independent of the operational and logistic parameters of the assigned units. These parameters are supplied by an interface to the database when the units are initially created. By separating the simulation from its data, any moveable unit can be modeled by customizing the database to reflect the desired unit with only minor modifications to the simulation. Once a campaign is in progress, the current state of the simulation can be saved to the database at any time, enabling easy resumption of the campaign. The user interface is constructed with the Java Swing

library to provide the familiar and ubiquitous Windows style environment without restricting the model to only operate in this environment. Having built the simulation in such manner, it is open to future expansion, such as a joint or coalition scenario with allies or new ship or aircraft platform configurations. These modifications can be accomplished by simply adding to the database with the specifications applicable to the added units and writing a small extension to the basic classes for those units.

NAVLOGS uses a discrete event simulation to drive the model rather than a time-step simulation. In a time-step simulation, the state of the system is updated at fixed time intervals, whether anything in the model has changed during that time interval or not. System processing time is needlessly wasted if the time interval is too small. If the time interval is too large, many events can occur in a single time step and must be adjudicated in a completely arbitrary manner. In contrast, the discrete event model only requires processing time to update the system state when events affecting the simulation occur. This discrete event paradigm was chosen since it captures the system state more accurately and more effectively. The simulation will now be addressed in greater depth.

B.   MODEL DESIGN

1.   Model Architecture

The components of NAVLOGS interact as shown in Figure 1. This structure, known as a "model-view-controller" design, separates the simulation model from the user interface. The design offers a great degree of flexibility for future enhancements and expansion. Since none of these components are rigidly tied together, any of them can be

12

easily replaced with little change to the others. For example, a new GUI can be constructed to improve interaction with the user without building a new non-visual simulation. Likewise, if more sophisticated models of the various units were desired, they can be readily incorporated into the simulation with no modification of the GUI required. The model-view-controller design is a key element to the extensibility of NAVLOGS.
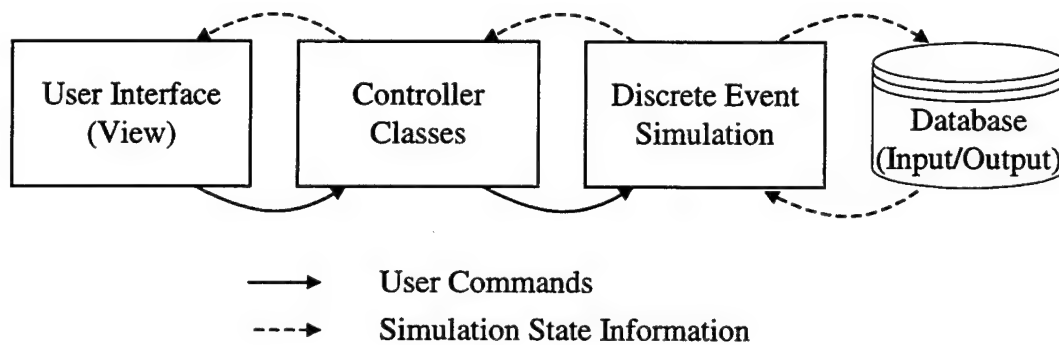

Figure 1. Model - View - Controller Architecture. The user interface is separated from the simulation by the controller classes. This allows either to be replaced or modified without changing the other. The data required by the simulation is stored and retrieved from a separate database.

The heart of the model is the discrete event simulation based on the Simkit package initially developed by Kirk Stork in [Ref. 4] and expanded by Professor Arnold Buss. The simulation is connected to the user interface via the controller classes. These controller classes pass information from the discrete event model, which is fundamentally non-visual, to the interface for display. Information is passed into the simulation from the database, to provide the necessary parameters to the model. Data from the model is also written back to the database for storage.

13

2. Class Organization

The modeled units used in the non-visual simulation are all Java objects sub-classed from Simkit BasicMover objects, as depicted in Figure 2. These objects maintain the state variables that define each unit's parameters, location, and movement characteristics at any given time in the simulation. These objects are further responsible for ensuring that infeasible values for these parameters are not assigned. They are extended from the BasicMover class to form a foundation class providing basic functionality for each type of platform. For example, the foundation class for all the ship objects is the class Ship, which provides the functionality common to all ships in the simulation. Examples of this basic functionality include methods to access and change state variables such as current F76 level and maximum speed. Other foundation classes include Aircraft, GroundVehicle, and AmphibiousVehicle.
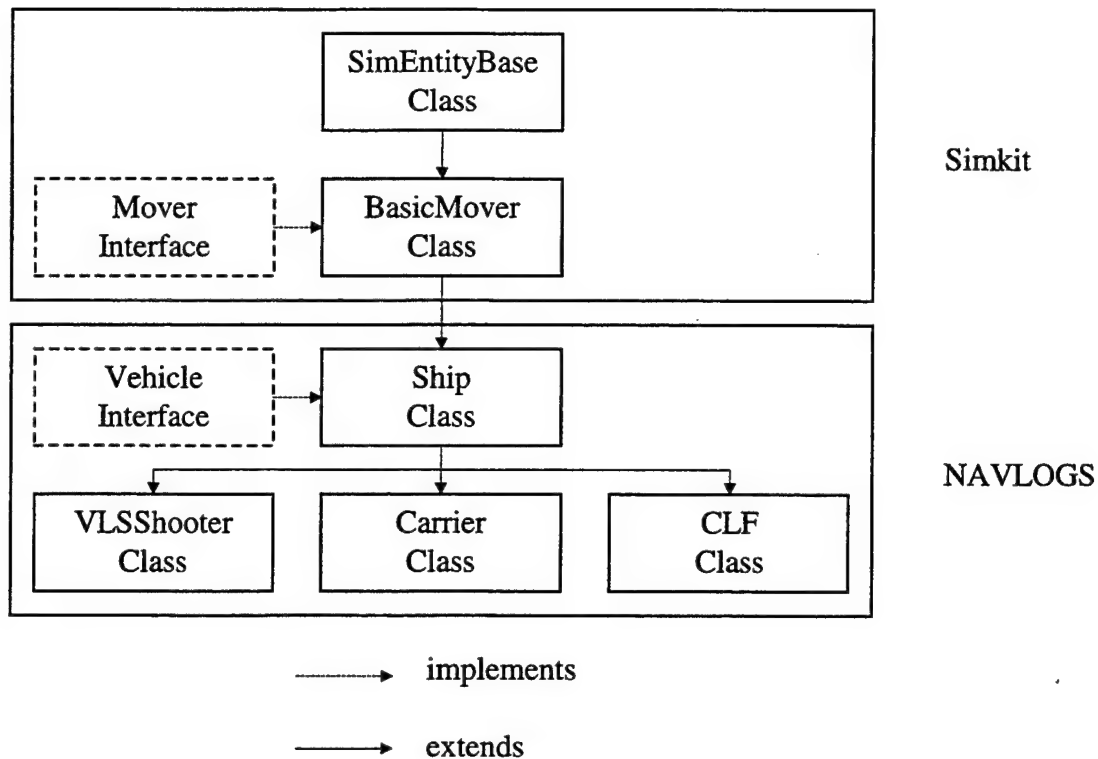
14

Figure 2. Sample Class Hierarchy Diagram. This diagram demonstrates the inheritance relationship of the Ship foundation class and its extended platform classes with Simkit.

A common feature of each foundation class is the implementation of the Vehicle interface, which requires the addition of two methods for data input and output. An example of the relationship of this implementation to the foundation classes is shown above in Figure 2. The first method, called `getProperties()`, returns a Java Properties object with key-value pairs of all dynamic state variables. The second method, called `getName()`, returns a unique name for each specific unit to identify its records in the database. These pieces of information are used in building Structure Query Language (SQL) statements to update the unit's information in the database. The

Properties object is also used by the GUI to present the current state of that unit to the user.

To maximize flexibility, specialization for specific platform types is contained in platform classes which sub-class the foundation classes. Examples of platform classes extended from the Ship class include Vertical Launching System (VLS) equipped escorts, Combat Logistics Force (CLF) ships, and aircraft carriers. These extended classes include only the additional functionality required to model the particular platform type desired, since they inherit all of the parameters and functionality of the Ship class. The class hierarchy diagram in Figure 2 shows the relationship of these platform classes to their foundation class, Ship. The computer code for the Ship and VLS Shooter classes and the Vehicle Interface can be found in Appendix B.

Because the platform classes inherit all the functionality of their foundation class, they are quite small since unnecessary replication of identical computer code has been avoided. This design yields another important advantage. If a function of a foundation class needs to be altered or a new function added, the modification only needs to be performed once. If each platform class explicitly included all the methods of its foundation class, modifying any function would have to be performed multiple times. Not only would this be needlessly repetitive, erratic behavior in the platform classes is possible if each modification is not identically performed.

Placing the specialized functionality for specific platform types in the smaller platform classes provides for easier expansion, as well as avoiding needless overhead. A

CLF ship does not need to keep track of its Tomahawk missiles, nor does a VLS escort need to maintain the state of its tactical aircraft inventory. Should the next generation of escorts make a radical change to weapon delivery, such as a purely hypothetical Horizontal Launching System ("HLS"), the model needs only to add another sub-class to the Ship class. With the new functions required for this weapon system added in the sub-class and a small change to the database (detailed below), these new "HLS" escorts can be easily integrated alongside existing units.

As outlined above, the Ship class provides the minimum functionality required to model a surface ship. Since logistics is the focus of this thesis, the state variables which track fuels and ammunition are critical. However, the Ship base class does not contain any variables for aircraft fuel or ordnance. The responsibility for ordnance is left for the sub-classes since weapon types and delivery systems vary considerably among platforms. Platform classes are also responsible for aviation fuel (JP-5), since its use also varies significantly between different types of ships. For example, with at most two helicopters, JP-5 is not a significant driver of logistics for the escort platforms, whereas an aircraft carrier consumes enormous quantities of JP-5 with even moderate flight operations. At the other extreme, jet fuel is merely cargo to the CLF platforms.

One of the most important functions of the Ship class is to maintain the current level of propulsion fuel. NAVLOGS uses the fuel consumption rates given in Schrady, Smyth, and Vassian [Ref. 5]. This work gives predictions for fuel consumption rates as a

17

function of ship speed for twenty-two classes of Navy ships. The predictions use a common function for all ship classes shown in (1). [Ref. 5: p 18]

$$Kgals / hour = b_0 + b_1 * e^{(b_2 (speed / 100)^3)} \qquad (1)$$

Each individual platform has unique values of b_0, b_1, and b_2 to yield its specific consumption rate. The propulsion fuel load is not continuously updated. Instead, each Ship maintains a time variable of when fuel level was last computed.

Unlike some dedicated combat models, units in this simulation simply cannot move without fuel. Each time speed or destination is changed, the fuel used since the last update is calculated and removed from the current level. When a Ship object is given a command to move or change speed, before the order is executed, a check is made to ensure that sufficient fuel remains to reach the destination. If the ship has sufficient fuel to travel to its destination, the appropriate events are scheduled (see Section 3, below). If not, the user is presented (via a dialog box) that there is not enough fuel, and the unit is stopped.

3. Movement Control

As described in the previous section, the Ship objects in the simulation maintain the parameters that describe their state at any given time. The task of actually moving them from point to point in the simulation is the job of a MoverManager class. The MoverManagers use the listener pattern of Simkit to track the units. When a MoverManager is instantiated, it registers itself as a listener of a Simkit mover. The Mover multicasts its events, such as beginning or ending a movement, to all registered

listeners. When the MoverManager "hears" the events of the Mover, it can issue appropriate commands to that Mover when they occur, as depicted in Figure 3. The listener pattern between the MoverManagers and Movers allows an existing manager to be used with any mover and likewise for new managers to be developed for existing Movers.
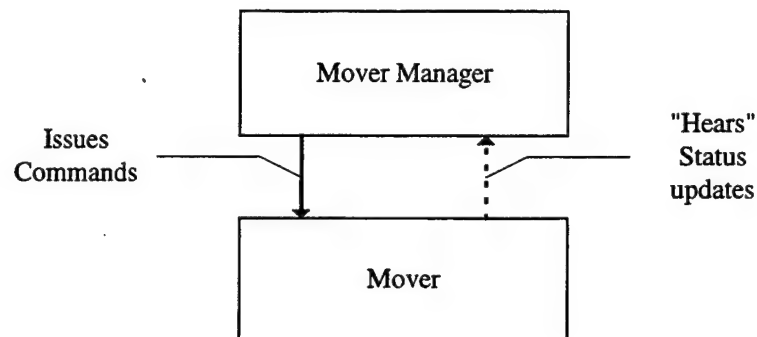

Figure 3. Generic MoverManager Listener Pattern. A Simkit Mover object is assigned to a MoverManager. The MoverManager is registered as a listener of the Mover, and can issue commands when the Mover multicasts its event occurrences.

In NAVLOGS, the UserCommandedMoverManager class is responsible for controlling the movement of Ships shown in Figure 4. Each of these managers is assigned a single Ship, with an initial destination and speed. When the simulation is started, the manager issues a command to its Ship to move to this destination at the ordered speed. The manager then listens for the Ship to reach the destination. When this occurs, the manager presents a dialog box to the user, requesting a new destination and speed. After user supplies a valid position, the process is started again.

19

```
                    ┌─────────────────────┐
                    │    UserCommanded     │
                    │    MoverManager      │
                    │                     │
 Issues    ─────────┤                     ├───────── "Hears"
 Commands           │    ▼            ▲    │           Status
                    └────│────────────│───┘           updates
                         │            ┊
                    ┌────▼────────────┊───┐
                    │                     │
                    │                     │
                    │        Ship         │
                    │                     │
                    └─────────────────────┘
```
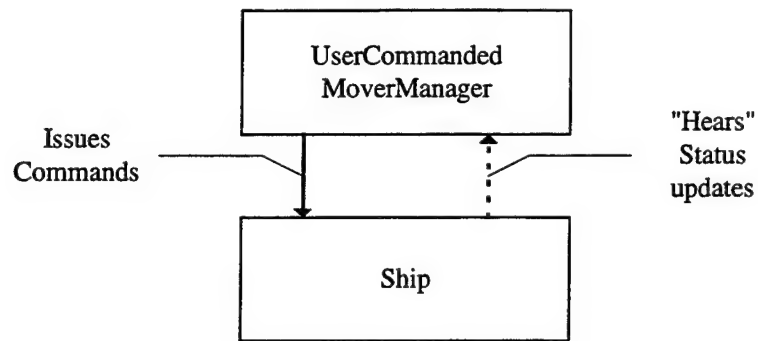
Figure 4. UserCommandedMoverManager Listener Pattern. A Ship object is assigned to a UserCommandedMoverManager that is registered as a listener of that Ship instance. When the UserCommandedMoverManager "hears" the Ship's EndMove event, it informs the user (via a dialog box) of the Ship's current position and requests a new destination and speed.

With the discrete event model of this simulation, there are only two events of interest in the course of moving a unit from a starting point to a destination. Assuming movement at a constant speed with no acceleration or deceleration considered, the unit starts moving at some time in the simulation. It arrives at the destination after an elapsed time equal to the distance divided by its speed. In Simkit, this elapsed time is computed and the destination arrival event is scheduled as soon as the unit starts moving. This sequence is shown below in Figure 5. As an example, consider a unit which must travel one hundred nautical miles at a speed of ten knots. If a time-stepped simulation were set to update only every hour, this movement would require the unit to be updated ten times.
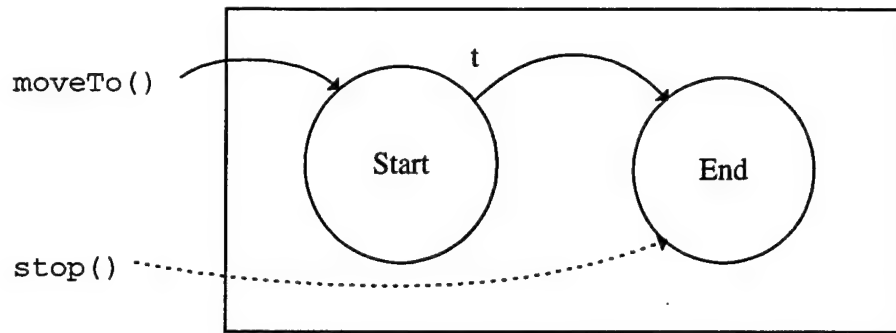
Figure 5. Movement Event Graph. When the `moveTo()` method is executed for a Mover, Simkit calculates the time required to reach the destination, and schedules an EndMove event with this time delay. The EndMove event can be preempted with the Mover's `stop()` method.

This simple uniform motion is sufficient for most purposes. However, if a more sophisticated movement model is required, the discrete event methodology is easily modified to accommodate more complex motion.

4.   Environmental Considerations

The primary focus of NAVLOGS is operational logistics; it is not intended to be a high-resolution combat simulation. However, since the targeted users of the simulation are predominately junior naval officers, a certain degree of realism is desired. A "cookie cutter" sensor and a flat earth model govern the basic interaction of targets and sensors provided by Simkit. The "cookie cutter" sensor is a circular area with radius equal to the maximum range of the sensor. As soon as a target is within the maximum range of the sensor, it is detected instantly. This type of sensor coupled with the flat earth model yields an undesired behavior for the purposes of this model. As an example, an air search radar may have a maximum range of hundreds of nautical miles, but it physically cannot detect a surface contact at that range due to the curvature of the earth. The reference

21

implementation in Simkit will allow this detection to occur. To circumvent these types of phenomenona and provide the desired degree of realism, NAVLOGS implements two improvements to Simkit's implementation. Figure 6, below, illustrates the geometry of the detection sequence described next.

Legend
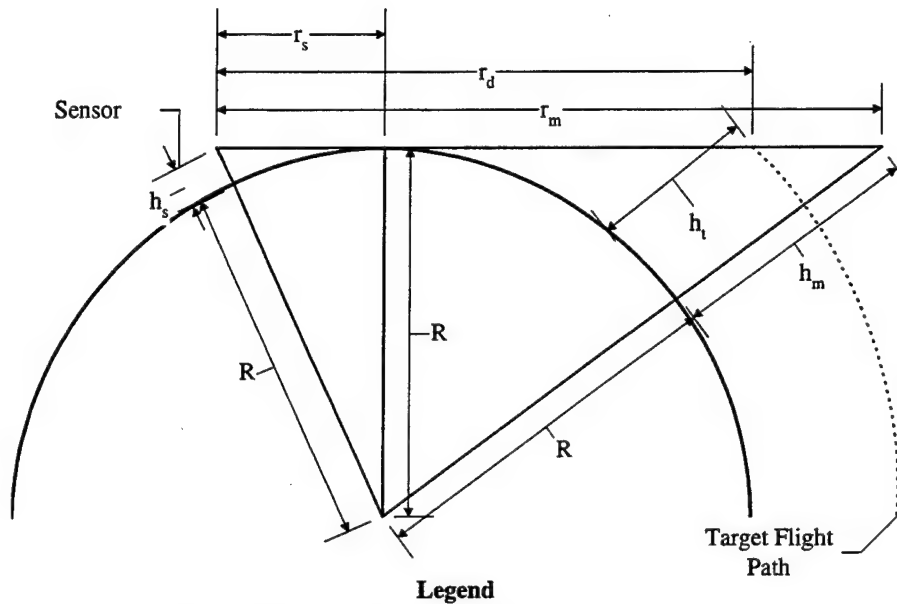
(All dimensions are measured in nautical miles)

h_m = height of the line of sight at maximum range

h_s = height of the sensor

h_t = height of the target

r_d = detection range of the target

r_m = maximum range of the sensor

r_s = sensor horizon range

R = Radius of the earth

Figure 6. Detection Sequence Diagram. This diagram demonstrates the geometry governing the interaction of a target and sensor as the target enters the range of the sensor.

22

First, as a target enters the maximum range of a sensor in the default implementation of Simkit, an EnterRange event is generated. This EnterRange event then immediately schedules a Detection event. NAVLOGS provides the CurvatureMediator class to delay the Detection event for a more realistic appearance. In adjudicating the behavior of sensors and targets, this class takes into account the curvature of the earth to prevent unrealistic detections like the example above.

Each friendly, neutral, and enemy unit in NAVLOGS has a height or altitude parameter. When the EnterRange event occurs, the CurvatureMediator computes sensor's horizon range, r_s, as shown in (2) using the radius of the earth, R, and the sensor's height, h_s, and subtracts this from the maximum range, r_m.

$$r_s = \sqrt{(R+h_s)^2 - R^2} \tag{2}$$

This is then used to compute the height of the line of sight at the maximum range, h_m, as shown in (3) and then compares this to the target's height, h_t, shown above in Figure 6.

$$h_m = \sqrt{(r_m - r_s)^2 + R^2} - R \tag{3}$$

If h_t is greater than h_m, then a Detection event is scheduled with no delay, since the target should be visible to the sensor. If the target's height or altitude is below the line of sight, further computations are required to determine when the Detection event should occur. The event graph in Figure 7 shows this progression.
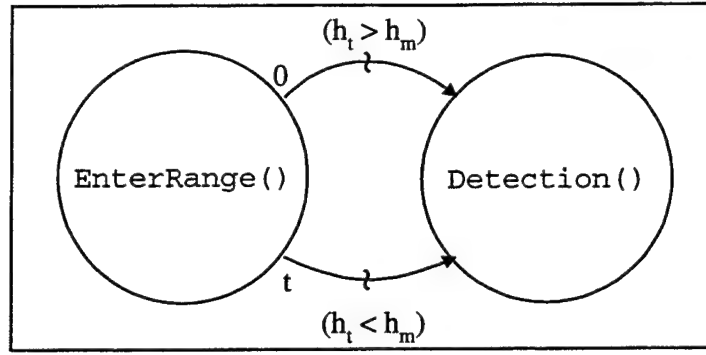
23

Figure 7. Target Detection Event Graph. When a target enters the range of a sensor, its height, h_t, is compared with the height of the line of sight, h_m, at that range. If h_t is greater than h_m, a Detection event is scheduled with no delay. Otherwise, the event is scheduled with a time delay equal to the distance the target must travel until h_t is equal to height of the line of sight divided by the relative speed of the target toward the sensor.

Once the CurvatureMediator determines that the target is not immediately detectable, it calculates the maximum theoretical detection range of the target, r_d, using (4).

$$r_d = \sqrt{(R+h_t)^2 - R^2} + \sqrt{(R+h_s)^2 - R^2} \tag{4}$$

This is the distance from the height of the line of sight at the sensor to the height of the line of sight equal to the target's altitude. This range is subtracted from the maximum range to determine the distance the target must travel before it is no longer shielded by the curvature of the earth. The Detection event is then scheduled after a time delay equal to this distance divided by the relative speed of the target toward the sensor. Although simple, this approach enables the sensor target interactions in the model to capture the earth's curvature.

The second enhancement to the basic Simkit implementation seeks to model the fact that the theoretical maximum range of an electromagnetic sensor is rarely achieved.

24

A variety of environmental factors such as target aspect and airborne contaminants can affect the actual range at which a target is detected. Each EnterRange event described above is accompanied by a corresponding ExitRange event. Likewise, each Detection has a similar Undetection. In order to include environmental variability, the Detection time determined by the CurvatureMediator is further delayed by a random exponentially distributed time. The Undetection is then scheduled by drawing another random time and subtracting it from the actual time the target will exit the sensor's range or be shadowed by the earth. This modification simply prevents targets from automatically being detected as soon as they enter the maximum range of the sensor.

These two improvements to the default implementation of Simkit demonstrate the power of modular design. To achieve these improvements, only two new classes were written that simply add the desired behavior. Objects created from these classes were able to interact with all existing Simkit objects. Thus, the new functionality required no modification to the existing version of Simkit.

5.   Simulation Animation

While the discrete event paradigm described above is desirable from a modeling perspective, the interactive nature of NAVLOGS requires a smoothly animated presentation to the user. As a result of the modular design of this thesis, it is possible for the underlying non-visual model to run as a discrete event simulation while the user's view of that model appears to be time-stepped. The GUI is registered as a listener to a PingThread2 object as shown in Figure 8. The PingThread2 class just "pings" at

25

periodic, fixed time intervals. The length of the time interval can be adjusted to speed or slow the pace of the presentation. When the GUI hears these pings, it gets a reference to all units still operating in the simulation. After obtaining a reference to the units, the GUI then loops over each one, obtains its current location, and paints it on the screen using standard Naval Tactical Data System (NTDS) symbology.
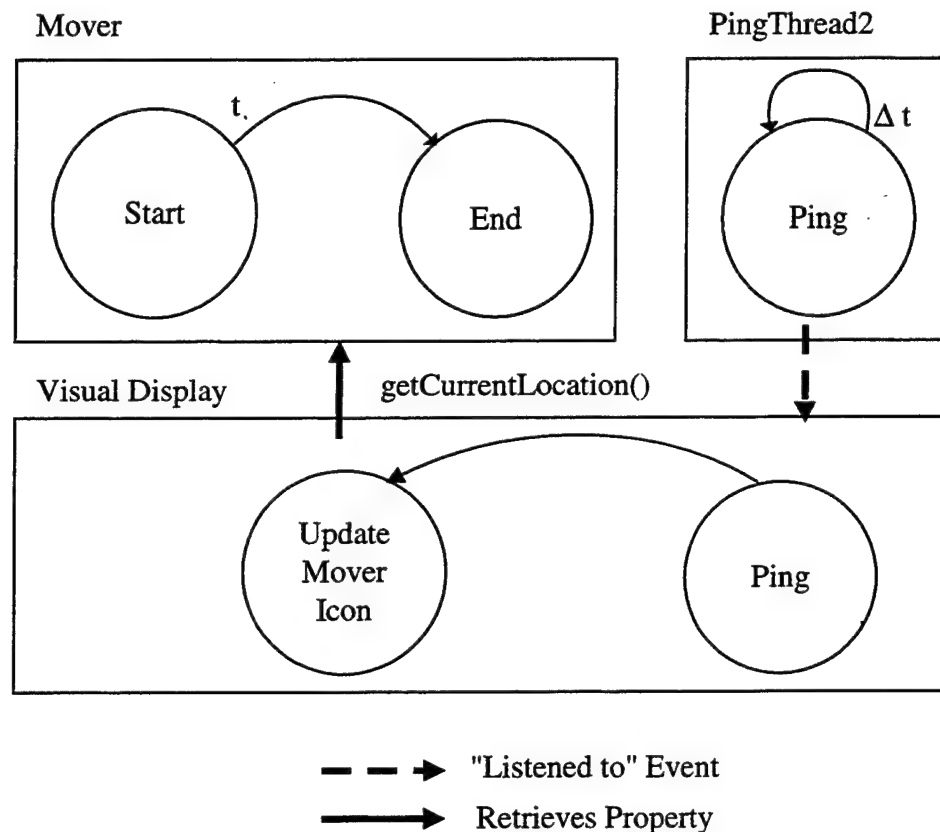
Figure 8. Animation Event Graph. The GUI is registered as a listener of the PingThread2 which "pings" at a fixed time interval, Δt. When the GUI "hears" the pings, it retrieves the current location of all Movers still in the simulation and updates their icons on the screen.

Each icon is a small image file that is contained in a Hashtable within the GUI.

There is no hard-coded connection between a unit and the NTDS symbol that is painted

on the screen. Hence, the icons could be easily changed without re-compiling any classes of the model. Each unit in the simulation maintains a state variable of its current classification, similar to the way contacts are maintained in a real, tactical environment. Examples of these classifications include friendly, neutral, unknown assumed enemy, and others. Each contact is further classified as a surface, sub-surface, air, or missile target. These classifications can change over time as more information about a contact is gathered. A low, slow flying helicopter provides a useful example of this. With a low altitude and slow speed, this helicopter may initially be designated as a neutral surface contact. If it then speeds up considerably or climbs, it will be re-designated as a neutral air contact. Finally, if intelligence is received that it is in the vicinity of a hostile installation or a friendly asset visually identifies it, it may be classified as an unknown assumed enemy air or even enemy air contact. Since the symbology is not rigidly fixed to each unit, it is possible for the symbol corresponding to each of these classifications to be shown to the user as the classification is updated. A sample screen showing an example of these symbols is shown in Figure 11 and Figure 12 in Chapter III.

C. BASIC SCENARIO DESCRIPTION

Like PROLOG, NAVLOGS will be equipped with an initial basic scenario. However, as previously noted, NAVLOGS will be flexible enough to enable easy reconfiguration through the use of the scenario editor developed by [Ref. 3]. The intended basic scenario consists of a notional battle group together with an Amphibious

Readiness Group (ARG). The ARG is supporting a Marine element operating ashore. A short discussion of the carrier battle group scenario follows.

The notional battle group is composed of a *Kitty Hawk* class carrier, two *Ticoderoga* class cruisers, two destroyers, one each from the *Arleigh Burke* and *Spruance* classes, and a *Supply* class fast combat support ship. The choice of a non-nuclear carrier is made to provide a greater requirement for F-76 fuel, increasing the strain on the logistics system. The available logistics assets will be further stressed with all VLS equipped escorts. Any missile re-supply for the escorts must be conducted in the forward logistics site (FLS), since there is no capability for underway replenishment of VLS launchers. A shuttle fleet consisting of a *Cimarron* class oiler and a *Kilauea* class ammunition ship operating from a forward logistics base are provided to re-supply the *Supply* class ship operating with the battle group.

Opposition to the user's battle group and ARG will be provided from computer controlled enemy air, surface, and sub-surface units. These units will interact with the friendly platforms in the non-visual model, driving controlling inputs from the user. Staff updates will prompt the user to conduct air strikes and engage enemy combat units as they are detected. As the user conducts these operations, fuel and ammunition resources for friendly surface and air units will be consumed at realistic rates as discussed in Section II.B.2, requiring the user to replenish these commodities with assigned logistic assets. The main purpose of the combat model is to provide consumption of fuel and ammunition, forcing the student to make realistic decisions to keep the force supplied.

28

In PROLOG, a student's performance is based on the accrual of points for damaging enemy units and targets, with corresponding point losses for damage to friendly units. When fully implemented, NAVLOGS will incorporate this performance system in addition to points based on the time averaged fuel and ammunition levels of assigned units. These criteria reflect the emphasis of NAVLOGS on operational logistics management, rather than combat operations.

The scenario editor developed by Lieutenant Sterba will allow the instructor to tailor the scenario to match improved weapons and platforms in the fleet. The scenario editor will also enable the instructor to effortlessly make modest changes in the scenario. [Ref. 3] NAVLOGS can therefore be tailored to meet the needs and goals of the logistics course. Since the composition of the fleet, the weapons it employs, and the missions it performs will undoubtedly change in the future, the scenario editor will further allow the simulation to remain current by modifying it without requiring a complete re-write. This basic scenario is a simple default example. Unlike PROLOG, additional scenarios in other parts of the world with completely different units can easily be loaded with no change to the computer code. The specifics of how a scenario is loaded into the simulation and saved once in progress will be addressed next in the following chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

III. DATA STORAGE AND RETRIEVAL

A. OVERVIEW

Data are input to the model through the use of two configuration files and an independent database. The configuration files are similar to the ".ini" files used by many applications in the Windows operating system. One file contains basic setup information about the graphical user interface (GUI), while the other holds information required to connect to the database. The implementation in this thesis connects to a Microsoft Access database through the Java JDBC-ODBC (Java Database Connectivity – Object Database Connectivity) bridge driver to store and retrieve the unit parameters for the model. The model will, however, operate without modification with any database that supports the structured query language (SQL) and has a platform Java Database Connectivity (JDBC) driver. The configuration files and database interface will next be examined in detail.

B. CONFIGURATION FILES DESCRIPTION

The initialization file consists of blocks of related information with a block title contained in braces (e.g. [Title]). Each block consists of parameter names on the left side of an equals sign and the corresponding value for the parameters on the right. (e.g. parameter1 = value). A sample versions of NAVLOGS' GUI configuration file may be found in Figure 9. The initialization file for the GUI contains three blocks of information. The first contains the parameters required for the initial display of the

31

window, including the title, location, and size. The paths to the graphics files for the scenario map and the symbology for the units are contained in the next block. The final block of this file lists the data controlling the speed of the animation.

```
[Window]
name = Naval Logistics Simulator
x = 0
y = 0
width = 800
height = 600
symbolHeight = 17
symbolWidth = 17

[Icons]
map = pictures/Bosnia1.jpg
FriendlySurface = pictures/FriendlySurface.gif
EnemySurface = pictures/EnemySurface.gif

[PingThread]
deltaT = 0.01
millisPerSimTime = 100
pinging = true
```

Figure 9. Sample NAVLOGS' GUI Configuration File. The first block contains information required to initially display the window. The second contains paths the graphics files for the map and unit icons. The final block contains the parameters controlling the speed of animation.

The file containing the database connection information is organized into two blocks, as shown in Figure 10. The first contains the name of the driver Java will use to connect to the database and the name and location of the database. The specifics of this connection will be examined in further detail in the next section. The second block of this file consists of the names of individual units in a given scenario and the names of the platform classes used to create those units. The unit names and the platform class names are used to create the initial SQL statements used to retrieve the saved parameters for

32

each unit and to locate a unit's record in the database for update. The advantage of using this input method for initialization information is that significant changes in the behavior and appearance of the simulation can be made without re-compiling any of the model computer code. As an example, in the two sample screens shown below in Figure 11 and Figure 12, the only difference is the name of the map file in the GUI initialization file.

```
[Database Info]
driverName = sun.jdbc.odbc.JdbcOdbcDriver
url = jdbc:odbc:navlogsData

[Units]
DDG-53 = VLSShooter
CG-65 = VLSShooter
CG-70 = VLSShooter
DD-973 = VLSShooter
```

Figure 10. Sample NAVLOGS Configuration File. The two blocks in this file contain the database connection information and unit identification data.
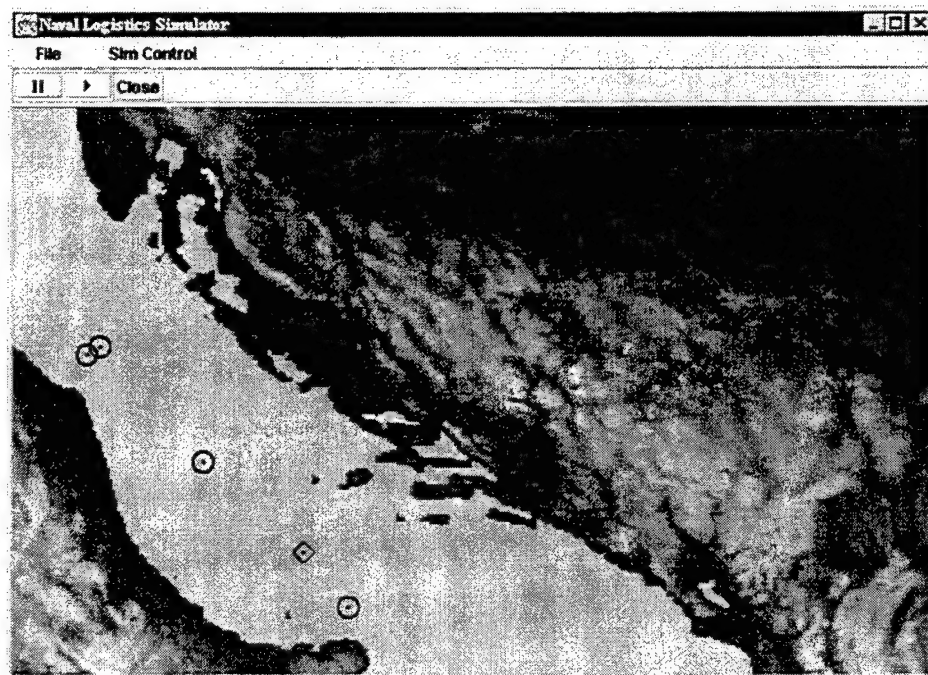

Figure 11. Sample Model Screen. This screen shot shows the GUI using the Bosnia1 map file in the initialization file.
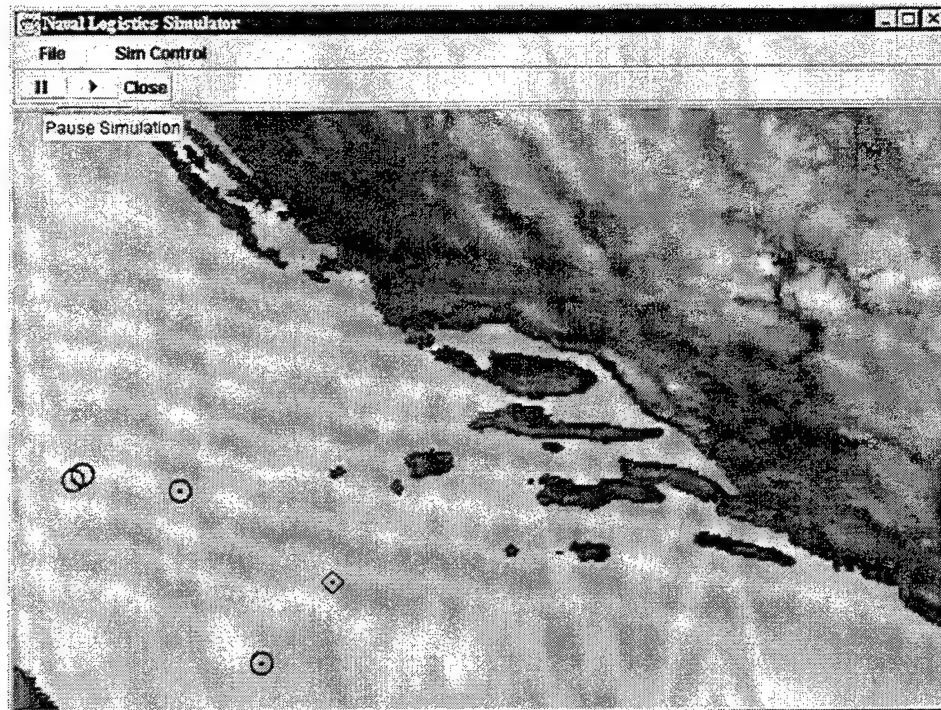
Figure 12. Sample Model Screen. This screen shot shows the GUI with the same simulation as Figure 11 and the Bosnia2 map file in the initialization file.

C. DATABASE DESCRIPTION

1. Database Communication

There are two ways a Java application can communicate with a database, as illustrated in Figure 13. Each mechanism uses the Java Database Connectivity (JDBC) application-programming interface (API) to communicate with the JDBC driver manager. The JDBC driver manager then can communicate with the database through a vendor or third party supplied JDBC driver, as shown on the right side of Figure 13. All commercially available databases like Oracle, Sybase, and many others have pure Java JDBC drivers available. Also, most freely-available database applications such as

Postgres, mSQL, and MySQL, likewise have JDBC drivers. Therefore, NAVLOGS can be easily configured to get data from most any database available. In the second communication path, shown on the left side of Figure 13, the JDBC-ODBC bridge connects to the database through the ODBC driver of the operating system.

This second method, required to connect to the Microsoft Access database used by this thesis, requires some system setup before a connection can be made. The system where the database file actually resides must have the 32 bit ODBC component installed. Next, an ODBC data source must be configured. The user specifies a data source name as shown in dialog box in Figure 14. This is the database name used in the initialization file to point the simulation to the database. Finally, under the Database section of the dialog, the user selects the database file containing the scenario information.
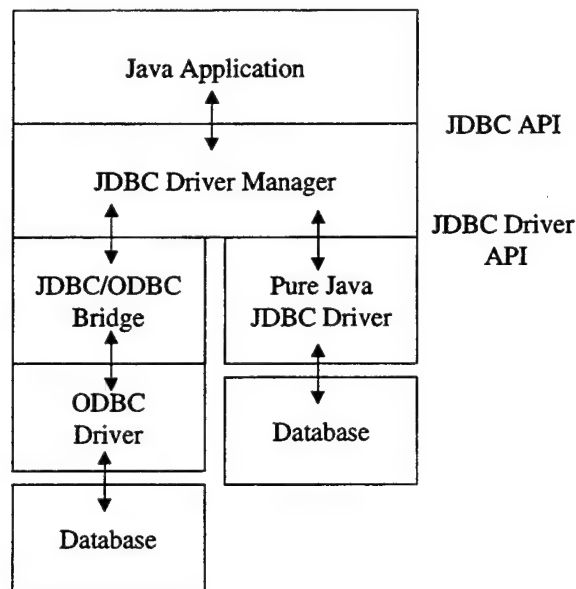

Figure 13. Possible Application to Database Communication Paths From Ref. 6: p 190. The path on the left is used to connect to an ODBC database; the right path demonstrates communication with any database having pure Java JDBC driver.
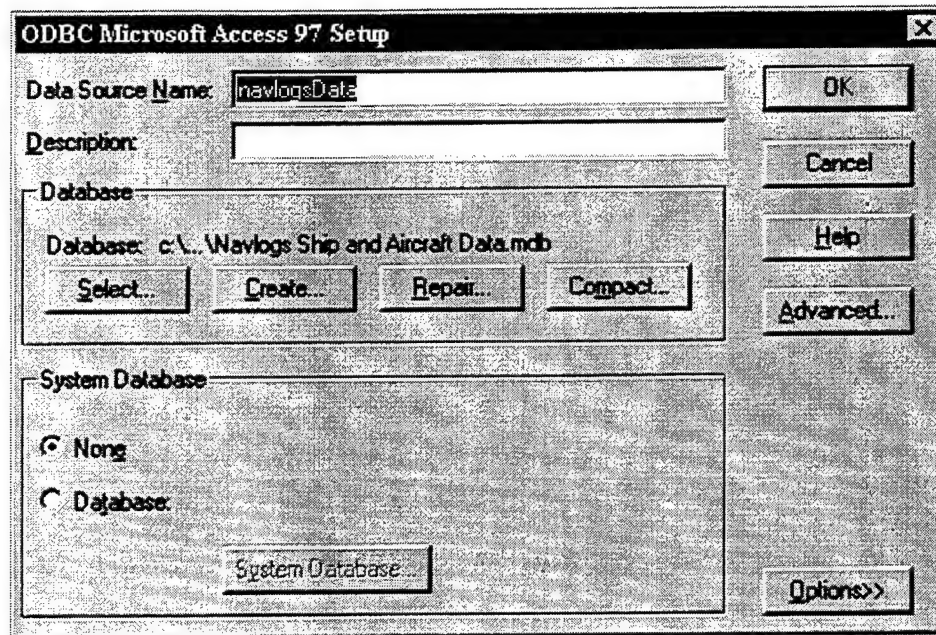
Figure 14. ODBC Data Source Configuration Dialog Box. The user first supplies a data source name and, optionally, a description. After selecting the database file and pushing the "OK" button, that database is available as an ODBC data source on the system.

2.   Database Structure

In the current version, the database consists of four different types of tables. Each base class is provided with a table to store the static setup information about the available platforms. There is a table with range information for the various sensors. The final two table varieties store the dynamic properties for the sub-classes and the MoverManagers. The database, working in concert with the initialization files, make the simulation independent of the data so that the same simulation computer code can run different scenarios with no modification or re-compilation.

The tables used for the base classes are intended to be used by the scenario editor to provide a listing of the available platform types that can be assigned to a given

36

scenario. A sample portion of a table for the Ship class is shown in Table 1. The scenario editor would read in the hullNumber column and use it to populate a menu of available Ships. The user selects a Ship and places it on the screen. The remaining information, along with the unit's position, is used to write a table for the extended classes to be accessed by the simulation. The sensor table is cross-referenced by these tables to provide maximum range information for each different sensor.

java Class	hull Number	surface Radar	air Radar	max Speed	F-76 Capacity	b0	b1	b2
VLSShooter	CG-47	SPS-67	SPY-1	32	597.0	-1429.04	2215.39	37.48
Carrier	CV-63	SPS-55	SPS-48	32	2197.2	-8937.6	10865.9	32.67
CLF	AOE-6	SPS-67		26	3740.4	12117.2	-12232.3	-25.79
Amphib	LHD-1	SPS-67	SPS-48	22	1850.0	-700.81	2039.41	78.21
VLSShooter	DD-963	SPS-55	SPS-40	33	492.0	-1812.92	3097.97	27.07

Table 1. Partial Foundation Class Database Table Showing Sample Static Parameter Values

Each sub-class has its own table, where the name of the table is the Java class name required to create a unit of that type. This name is also contained in the initialization file for each individual unit. This structure allows the constructed SQL statement to locate a unit's record in the database. Within the table, the columns are the different state parameters required for a unit of that type. The rows of the table are records for individual units of that type. A small sample is shown in Table 2. The javaClass entry in the table is used to identify what type of object will be instantiated.

37

NAVLOGS then uses an advanced Java feature called reflection to instantiate the objects and populate the instance variables.

javaClass	name	xCoord	yCoord	currentF76	vlsCells	sm2s
VLSShooter	CG-65	84.32	107.68	507.31	122	54
VLSShooter	CG-70	98.46	99.63	474.34	122	60
VLSShooter	DD-973	124.33	119.91	447.71	61	0
VLSShooter	DDG-53	102.41	77.35	389.85	90	40

Table 2. Partial Platform Class Showing Sample Dynamic Parameter Values

To add a new type of platform, the model requires only the new sub-class detailed in Chapter II and a new table added to the existing database named for the new extension. Finally, any units of this type to be added to an existing scenario need to be added to the units block of the initialization file with their names and the new class name. The final table type holds the MoverManager information. Table 3 shows a representation of the information for the units shown in Table 2.

name	destX	destY	current Speed
CG-65	100.0	10.0	15.0
CG-70	110.0	5.0	15.0
DD-973	90.0	5.0	15.0
DDG-53	100.0	0.0	15.0

Table 3. Sample MoverManager Table with Example Values

3.   Data Access

The task of reading data from the database and updating the dynamic tables is the responsibility of the DBInterface class.   This class provides a number of methods allowing it to read in unit records and create the objects of the simulation, as well as updating the units' information in the database when the user saves a scenario in progress. This class has no direct knowledge of either the simulation or the database until it is created. When the simulation is started, a DBInterface is created and given the database connection information from the initialization file. The individual unit names and the names of their sub-classes are placed in a Java Properties object. A reference to the newly created DBInterface is then passed to the GUI when it is created.

Once the GUI has a reference to the DBInterface, the GUI calls the loadUnits() method of DBInterface. This method loops over each unit in the Properties object described above and looks for a table in the database matching the sub-class name associated with that unit. If the search is successful, the unit's information is retrieved and used to instantiate that unit, and the unit is added to a Java Vector object. After the loop is completed, the Vector of units is returned to the GUI for display.

A similar method is used to update the status of the units in the simulation. The updateUnits() method of DBInterface accepts a Vector of units from the GUI. The method loops over the Vector, determines the sub-class and name of each unit, then looks in the database for a corresponding entry. If an entry is found, the method retrieves the

39

Properties object from the unit with the current values of its state variables, constructs an SQL statement with the updated information, and updates the record in the database.

The use of an independent database together with the configuration files greatly enhances the flexibility of the simulation. The parameters, which provide the desired behavior of the units in the simulation, can be easily changed to model other units with no change required to the computer code. Likewise, the composition of the forces in the simulation can also be quickly changed. This gives the simulation the power to run multiple scenarios or alter existing scenarios with little difficulty.

IV.   SUMMARY AND RECOMMENDATIONS

A.   THESIS RESULTS

Building an interactive logistics wargame to support the entire vision of NAVLOGS is a tremendous undertaking, far beyond the scope of a single master's thesis. While this thesis has not brought the project to the desired end state, it has provided a solid foundation on which further development can be built. The use of the Java programming language, modular design, and use of a separate database and configuration files enable this foundation to operate on many operating systems and even multiple computers.

The discrete event simulation is scalable, and expanded functionality can be added without extensive modification to the existing model. The simulation provides basic ships that move in an environment that captures the curvature of the earth. These ships consume fuel at accurate rates for their respective classes. More importantly, since the primary aim of this project is modeling naval operational logistics, the model's logic keeps ships from moving without sufficient fuel. The modular architecture enables the underlying model to operate as a discrete event simulation, while the presentation to the user appears to be a smoother looking, time-stepped model.

Through the use of a separate database and simple configuration files, the simulation is independent of its required data and parameters. This feature enables the simulation to run multiple scenarios with distinct units assigned and different maps or

41

symbology without re-compilation of the computer code. The use of JDBC drivers in conjunction with a network-accessible database can ultimately support a logistics wargame running on a local area network, or even the Internet. There are several areas for future work on this project in order to bring NAVLOGS to its full design concept.

B. RECOMMENDATIONS FOR FURTHER DEVELOPMENT

1. Simulation Modifications

The single most important work needed in the simulation is the addition of a combat model between the two sides. Currently, the friendly units are maneuvered by the user as described in Chapter II, while the enemy units simply follow a preset path. As they move in the simulation, units from the two sides detect one another, although the combat portion is not yet implemented. The next step will be to develop a combat model with realistic means of depleting ammunition, thus generating requirements for replenishment. A second key improvement to the current version of the simulation is the need to model both land and sea. In his thesis research, Lieutenant John Sterba has developed the necessary methods and algorithms to model this, but this necessary behavior has yet to be integrated with this thesis' work.

2. Data Storage and Retrieval

The configuration setup required to connect the Microsoft Access database outlined in the previous chapter can be made more convenient for the user. Within the current setup, the instructor is required to configure each different scenario's database in the Windows operating system. Using this type of database also fixes the database to

only one machine. By using a database application with a pure Java JDBC driver, the database file can be located anywhere. For example, the database could be located on a server over a network or the Internet. The only change required to access this new database would be to update the initialization file with the driver name, the URL to the database, and any logon information like user name or password. Since the DBInterface class has been written in a completely generic manner, it will still read and write correctly to any database to which it can connect.

To demonstrate this, the tables from the Microsoft Access database were imported into a database application called MySQL running on the Linux operating system. By changing the initialization files to point to a pure Java JDBC driver for MySQL, the current version of NAVLOGS was run without any other modification

3. Scenario Editor

The scenario editor developed by Lieutenant Sterba allows the instructor to add any map to a scenario and add the coastlines to prevent ships from moving on land and conversely, land vehicles from driving in the sea. The editor also allows units to be assigned on the map from a menu. Currently, the menu containing the various units is read from a text file. Integration of Lieutenant Sterba's scenario editor with the classes developed in this thesis is another important step in the process to fully implement NAVLOGS. To integrate the scenario editor into this thesis, it needs only to add the necessary methods to the DBInterface class to populate the units menu from the database,

43

read the static setup information for the assigned unit, and write that information with the position back to the appropriate table in the database.

APPENDIX A. OBJECT ORIENTED PROGRAMMING AND JAVA

This appendix provides a basic introduction to the fundamentals of Object Oriented Programming (OOP) and Java. If the reader desires more in depth coverage of the subject, many excellent references, such as [Ref. 7], are available.

An object-oriented language such as Java differs from procedural programming languages such as Pascal. A procedural language consists of data structures together with algorithms that manipulate the data. In OOP, the primary data structure is the object. Each object is defined by a class, which is "...a container for the data and methods (functions) that make up part or all of an application." [Ref. 7: p 52] Thus, in OOP, the data and algorithms are bound together in the objects. An OOP can be compared with the parts of the modern personal computer (PC). Just as the PC is assembled from many replaceable components that have certain functionality, an OOP program is assembled from collections of objects. [Ref. 7: pp. 104-5]

In Java the properties of an object are contained in its instance variables. These variables may be numbers, strings of text, or even other objects. Taken together the values of these variables describe the state of the object at any given time. The instance variables of an object are hidden from other objects to prevent inadvertent or malicious modification by declaring them to be "private". Other objects can gain access to the private instance variables of an object only through its methods. This powerful feature of OOP is known as *encapsulation.* [Ref. 7: p107]

> Formally, encapsulation is nothing more than combining data and
> behavior in one package and hiding the implementation of the data from

45

the user of the object.... Encapsulation is the way to give the object its "black box" behavior, which is the key to reuse and reliability. Since this means an object may totally change how it stores its data but, as long as it continues to use the same methods to manipulate the data, no other object will know or care. [Ref. 7: p 107]

The two most common types of methods are the accessor and mutator, often-called "getters" and "setters". As the name implies getters retrieve the values of instance variables, and setters modify them.

A second important feature of OOP is the notion of *inheritance*. Simply put, this means that a class can build upon or extend another class. The class that is extended is commonly called the superclass or parent class, while the extending class is most often referred to as the sub-class or child class [Ref 7: p 156]. The child class "...initially has all the properties and methods of its parent." [Ref. 7: p 106] Like any other class, though, a child must access the private instance variables of its parent through the parent's methods. The child may override any or all methods of its parent to easily modify its behavior. This leads to the principle of *polymorphism*. This principle is explained in the following quotation.

When you send a message that asks a subclass to apply a method using certain parameters, here is what happens:
- The subclass checks whether or not it has a method with that name and with *exactly* the same parameters. If so, it uses it.

If not,

- Java moves to the parent class and looks there for a method with that name and those parameters. If so, it calls that method

Since Java can continue moving up the inheritance chain, parent classes are checked until the chain of inheritance stops or until Java finds a matching method. [Ref. 7: p 162]

An object is created or instantiated from a class using the *new* keyword. The new object is said to be an instance of that class. Several events occur when an object is instantiated in Java. System memory is set aside for the new object, and a special method is executed. The constructor method of the object receives any initial values of the instance variables and assigns them. It also conducts any required additional setup for the object, such as registering the object with other objects in the application. A powerful feature of Java over other programming languages is the garbage collector. When an object is no longer required by the application, the garbage collector reclaims the memory that was used by that object and makes it available to the system again.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. SAMPLE Computer CODE

```java
/**
 *   navlogs.smd.Vehicle Interface
 *
 *   @author Anthony Troxell
 **/

package navlogs.smd;

import java.util.*;
import simkit.*;
import simkit.smd.*;

public interface Vehicle extends Mover {

//instance methods

    /**
     *   A required override, since the SimEntityBase method appends a
     *   serial.  Without the base name the units database record will not
     *   be found when attempting to update.
     **/
    public String getName();

    /**
     *   Required for a NAVLOGS unit to build the SQL update statements to
     *   save its current state.
     **/
    public Properties getProperties();

}
```

```java
/**
 *   navlogs.smd.Ship Class
 *
 *   A generic class for surface ship units.
 *
 *   @author Anthony Troxell
**/

package navlogs.smd;

import java.util.*;
import javax.swing.*;
import navlogs.utility.*;
import simkit.*;
import simkit.smd.*;

public class Ship extends BasicMover implements Vehicle{

// instance variables

    private boolean uSA = true;
    private boolean unrepStatus;
    private boolean zigStatus;
    /**
     *   The three "b" variables are used to determine fuel consumption
     *   rates for various speeds in the following formula:
     *   Kgals/hour = b0 + b1 * exp( b2 * ( speed / 100 ) ^ 3 )
    **/
    private double b0;
    private double b1;
    private double b2;
    private double currentF76;
    private double maxF76;
    private double maxSpeed;
    private double timeLastFuelUpdate;
    private ImageIcon symbol;
    private Properties stateVars;
    private String name;

// constructor methods

    public Ship(Properties prop){
        super(getBasicMoverInfo(prop));
        stateVars = (Properties) prop.clone();
        try{
            uSA = Utility.bool(stateVars.get("uSA"));
            unrepStatus = Utility.bool(stateVars.get("unrepStatus"));
            zigStatus = Utility.bool(stateVars.get("zigStatus"));
            b0 = Utility.doub(stateVars.get("b0"));
            b1 = Utility.doub(stateVars.get("b1"));
            b2 = Utility.doub(stateVars.get("b2"));
            currentF76 = Utility.doub(stateVars.get("currentF76"));
            maxF76 = Utility.doub(stateVars.get("maxF76"));
            maxSpeed = Utility.doub(stateVars.get("maxSpeed"));
            timeLastFuelUpdate =
              Utility.doub(stateVars.get("timeLastFuelUpdate"));
            name = stateVars.get("name").toString();
```

50

```java
        /**
         *   None of the "b" variables or the unit name are
         *   subject to change after instantiation so they are
         *   removed from the Properties.
         **/
        stateVars.remove("b0");
        stateVars.remove("b1");
        stateVars.remove("b2");
        stateVars.remove("name");
    }
    catch(NullPointerException e){System.out.println(e +
     ": Invalid Properties object to create Ship.");
    }
}
protected static String getBasicMoverInfo(Properties prop){
    StringBuffer buf = new StringBuffer();
    buf.append(prop.get("name") + " ");
    buf.append(prop.get("xCoord") + " ");
    buf.append(prop.get("yCoord")+ " ");
    buf.append(prop.get("maxSpeed"));
    return buf.toString();
}

// instance methods

public void addCurrentF76(double newF76){
    if(currentF76 + newF76 <= maxF76 && currentF76 + newF76 >= 0){
        currentF76 += newF76;
    }
    else if(currentF76 + newF76 < 0){
        currentF76 = 0;
    }
    else{
        currentF76 = newF76;
    }
}
/**
 *   This method first removes the fuel used since the last update,
 *   updates the time of last update to the current simTime, then
 *   returns the amount of fuel remaining.
 **/
public double getCurrentF76(){
    double timeElapsed = Schedule.simTime() - timeLastFuelUpdate;
    this.addCurrentF76(-(timeElapsed * getFuelUseRate()));
    timeLastFuelUpdate = Schedule.simTime();
    return currentF76;
}
public boolean isUNREP(){
    return unrepStatus;
}
public boolean isUSA(){
    return uSA;
}
/**
 *   Used by the getCurrentF76 and setSpeed methods to calculate the
 *   rate of fuel use since the fuel level update.
```

51

```java
**/

private double getFuelUseRate(){
    return (b0 + b1 * Math.exp(b2 * (Math.pow(this.getSpeed() / 100,
    3)))) / 1000;
}
/**
 *  Used by the moveTo method to calculate the rate of fuel use for
 *  the intended move.
 *  @param atSpeed the speed of the next movement
**/
private double getFuelUseRate(double atSpeed){
    return (b0 + b1 * Math.exp(b2 * (Math.pow(atSpeed / 100, 3)))) /
    1000;
}
public double getMaxF76(){
    return maxF76;
}
public double getMaxSpeed(){
    return maxSpeed;
}
/**
 *  Implements Vehicle.
**/
public String getName() {
    return name;
}
/**
 *  All state variables which can be changed over the course of the
 *  simulation are updated in the stateVars Properties object and a
 *  copy of the Properties object is returned to build the update SQL
 *  statement.
 *
 *  Implements Vehicle.
**/
public Properties getProperties(){
    stateVars.put("uSA", String.valueOf(uSA));
    stateVars.put("unrepStatus", String.valueOf(unrepStatus));
    stateVars.put("zigStatus", String.valueOf(zigStatus));
    stateVars.put("currentF76", String.valueOf(currentF76));
    stateVars.put("maxF76", String.valueOf(maxF76));
    stateVars.put("maxSpeed", String.valueOf(maxSpeed));
    stateVars.put("timeLastFuelUpdate",
     String.valueOf(timeLastFuelUpdate));
    stateVars.put("xCoord",
     String.valueOf(this.getCurrentLocation().getXCoord()));
    stateVars.put("yCoord",
     String.valueOf(this.getCurrentLocation().getYCoord()));
    return (Properties) stateVars.clone();
}
public ImageIcon getSymbol(){
    return symbol;
}
public double getTimeLastFuelUpdate(){
    return timeLastFuelUpdate;
}
public boolean getZigZag(){
```

```java
        return zigStatus;
    }


    /**
     *  Order the BasicMover to begin moving to the given destination at
     *  the maximum speed.
     *  @param destination The current destination of the Ship.
     **/
    public void moveTo(Coordinate newDestination) {
        this.moveTo(newDestination, maxSpeed);
    }
    /**
     *  Order the BasicMover to begin moving to the given destination at
     *  the given speed (or the maximum speed, whichever is smaller).
     *  This method overrides the moveTo method of BasicMover to check if
     *  there is sufficient fuel for the Ship to travel to the
     *  destination.
     *  @param destination The current destination of the Ship.
     *  @param atSpeed The speed at which the BasicMover is to travel or
     *         maxSpeed, whichever is smaller).
     **/
    public void moveTo(Coordinate newDestination, double atSpeed) {
        double time = newDestination.distanceFrom(
          this.getCurrentLocation()) / atSpeed;
        if(time * this.getFuelUseRate(atSpeed) < this.getCurrentF76()){
            setDestination(newDestination);
            setSpeed(atSpeed);
            waitDelay("StartMove", 0.0, this);
        }
        else{
            JOptionPane.showMessageDialog((JFrame)null, "Insufficient fuel
              to " + "reach " + newDestination, "Low Fuel Warning " +
              this.getName(), JOptionPane.WARNING_MESSAGE);
        }
    }
    public void setIsUNREP(boolean isUNREP){
        unrepStatus = isUNREP;
    }
    /**
     *   This method allows for a reduction in max fuel due to battle
     *   damage.
     **/
    public void setMaxF76(double newMaxF76){
        maxF76 = newMaxF76;
    }
    /**
     *   This method allows for a reduction in max speed due to battle
     *   damage.
     **/
    public void setMaxSpeed(double speed){
        maxSpeed = speed;
        if(this.getSpeed() > maxSpeed){
            super.setSpeed(maxSpeed);
        }
    }
    /**
     *  This method first removes the fuel used since the last update,
```

```java
 *   updates the time of last update to the current simTime, then sets
 *   the unit's speed.
 *   @param newSpeed the new speed of movement
**/
public void setSpeed(double newSpeed){
    double timeElapsed = Schedule.simTime() - timeLastFuelUpdate;
    this.addCurrentF76(-(timeElapsed * getFuelUseRate()));
    timeLastFuelUpdate = Schedule.simTime();
    super.setSpeed(Math.min(newSpeed, maxSpeed));
}
public void setSymbol(ImageIcon symbol){
    this.symbol = symbol;
}
public void setZigZag(boolean zigZag){
    zigStatus = zigZag;
}
public String toString(){
    StringBuffer buf = new StringBuffer();
    buf.append(super.toString() + "\n");
    if(isVerbose()){
    //  These lines are for troubleshooting.
        buf.append("Current F76: " + currentF76 + "\n");
        buf.append("Max F76: " + maxF76 + "\n");
        buf.append("Last Fuel Update: " + timeLastFuelUpdate + '\n');
        buf.append("b vals: " + b0 + " " + b1 + " " + b2 + "\n");
        buf.append("ZigZag: " + zigStatus + "\n");
        buf.append("USA: " + uSA + "\n");
        buf.append("unrep Status: " + unrepStatus + "\n");
    }
    return buf.toString();
}
}
```

```java
/**
 *   navlogs.smd.VLSShooter Class
 *
 *   Extends the Ship class to model VLS Launcher equipped missile
 *   escorts.
 *
 *   @author Anthony Troxell
 **/

package navlogs.smd;

import java.util.*;
import navlogs.utility.*;
import simkit.*;
import simkit.smd.*;

public class VLSShooter extends Ship{// implements Vehicle{

// instance variables

    private int sm2s;
    private int thawks;
    private int vlsCells;
    private Properties stateVars;

// constructor methods

    public VLSShooter(Properties prop){
        super(getShipInfo(prop));
        sm2s = Utility.intg(prop.get("sm2s"));
        thawks = Utility.intg(prop.get("thawks"));
        vlsCells = Utility.intg(prop.get("vlsCells"));
        stateVars = new Properties();
    }
    protected static Properties getShipInfo(Properties prop){
        Properties shipProp = (Properties) prop.clone();
        shipProp.remove("vlsCells");
        shipProp.remove("sm2s");
        shipProp.remove("thawks");
        return shipProp;
    }

// instance methods

    public Properties getProperties(){
        stateVars = super.getProperties();
        stateVars.put("sm2s", String.valueOf(sm2s));
        stateVars.put("thawks", String.valueOf(thawks));
        stateVars.put("vlsCells", String.valueOf(vlsCells));
        return (Properties) stateVars.clone();
    }
```

55

```java
    public String toString(){
        StringBuffer buf = new StringBuffer();
        buf.append(super.toString());
        if(isVerbose()){
            // these lines are for troubleshooting
            buf.append("VLS Cells: " + vlsCells + "\n");
            buf.append("SM-2's: " + sm2s + "\n");
            buf.append("Tomahawks: " + thawks + "\n");
        }
        return buf.toString();
    }
}
```

LIST OF REFERENCES

1. Miller, Marvin O., ed., *Underway Replenishment of Naval Ships,* Underway Replenishment Department, Port Hueneme Division Naval Surface Warfare Center, Port Hueneme, CA, 1992.

2. Mitchell, Mark L., *PRO-LOG Player's Manual,* Operations Research Department, Naval Postgraduate School, Monterey, CA, 1988.

3. Sterba, John, *Operational Maneuver from the Sea Logistics Training Aid,* Masters Thesis, Operations Research Department, Naval Postgraduate School, Monterey, CA, 1999.

4. Stork, Kirk A., *Sensors in Object Oriented Discrete Event Simulation,* Masters Thesis, Operations Research Department, Naval Postgraduate School, Monterey, CA, 1996.

5. Schrady, David A., Gordon K. Smyth, and Robert B. Vassian, *Predicting Ship Fuel Consumption: Update,* Operations Research Department, Naval Postgraduate School, Monterey, CA, 1996.

6. Horstmann, Cay S. and Cornell, Gary, *Core Java 1.1, Volume II, Advanced Features,* Sun Microsystems Press, Mountain View, CA, 1998.

7. Horstmann, Cay S. and Cornell, Gary, *Core Java 1.1, Volume I, Fundamentals,* Sun Microsystems Press, Mountain View, CA, 1997.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center..2
   8725 John J. Kingman Rd., STE 0944
   Fort Belvoir, VA 22060-6218

2. Dudley Knox Library..2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Defense Logistic Studies Information Exchange..1
   U.S. Army Logistics Management Center
   Fort Lee, VA 23801-6043

4. Deputy Chief of Naval Operations (Logistics)..1
   Attn: CDR Carolyn Kresek, N421C
   2000 Navy Pentagon
   Washington, DC 20350-2000

5. Professor David Schrady, Code OR/So..1
   Department of Operations Research
   Naval Postgraduate School
   Monterey, CA 93943-5000

6. Professor Arnold Buss, Code OR/Bu..1
   Department of Operations Research
   Naval Postgraduate School
   Monterey, CA 93943-5000

7. CDR Kevin Maher, Code OR/Mk..1
   Department of Operations Research
   Naval Postgraduate School
   Monterey, CA 93943-5000

8. LT Anthony Troxell..1
   818 Tharp St, #106
   Arlington, TX 76010